CPS

Last Update: 2025-03-17

Contents

零、Acknowledgement

本书[1]所描述的编译器——New Jersey 版本的 Standard ML（SML/NJ）——是众多开发者共同努力的成果。David B. MacQueen 和我于 1986 年开始了这一项目，最初的计划是在大约一年的时间内开发一个 Standard ML 的前端，以作为进一步研究的工具。David 主要负责类型检查器和模块系统的开发；我们共同完成了解析器、抽象语法、静态环境机制以及语义分析的工作。而我则主要关注动态语义、中间表示、优化、代码生成以及运行时系统的设计与实现。

当然，最终我们在这个项目上的投入远超最初的预期，耗时超过五年。这是因为项目的目标变得更加宏大：构建一个完整、健壮、高效且可移植的 Standard ML 编程环境。如果没有众多才华横溢的开发者的帮助，我们不可能完成这一工作。按照字母顺序，他们是：

Bruce F. Duba 协助改进了模式匹配编译器、CPS 常量折叠阶段、内联展开阶段、寄存器溢出处理阶段以及编译器的许多其他部分。此外，他还参与了 `call with current continuation` 机制的设计。

Adam T. Dingle 实现了调试器的 Emacs 模式。

Lal George 为代码生成器添加了对浮点寄存器的支持，并显著提升了浮点运算的性能。此外，他还修复了由我和其他人引入的一些棘手的错误。

Trevor Jim 参与了 CPS 表示的设计，并实现了模式匹配编译器和闭包转换阶段，以及最初的浮点函数库和外部原始函数的汇编语言实现。

James S. Mattson 实现了用于构建编译器的首个词法分析器生成器。

James W. O'Toole 实现了 NS32032 代码生成器。

Norman Ramsey 实现了 MIPS 代码生成器。

John H. Reppy 对运行时系统进行了多次改进和重写。他实现了信号处理机制，优化了 `call with current continuation` 机制，设计了当前用于调用 C 语言函数的机制，并实现了一套先进的新垃圾回收器，使运行时系统更加健壮。此外，他还实现了 SPARC 代码生成器。

Nick Rothwell 协助实现了独立编译机制。

Zhong Shao 实现了 Common Subexpression Elimination（公共子表达式消除），并设计了使用 multiple-register continuations（多寄存器续体）以加速过程调用的 callee-save（被调用者保存）约定。

David R. Tarditi 改进了词法分析器生成器，并实现了用于构建前端部分的解析器生成器；他还协助实现了调试器使用的类型重构算法。此外，他与 **Anurag Acharya** 和 **Peter Lee** 共同实现了 ML 到 C 的翻译器。

Mads Tofte 协助实现了 separate compilation mechanism（分离编译机制）。

Andrew P. Tolmach 实现了 SML/NJ 调试器。此外，他还以更加函数式的风格重写了静态环境（符号表）。

Peter Weinberger 实现了首个版本的 copying garbage collector（复制式垃圾回收器）。

最后，我要感谢那些在本书早期草稿阶段提供了宝贵意见的同行：**Ron Cytron**、**Mary Fernandez**、**Lal George**、**Peter Lee**、**Greg Morrisett**、**Zhong Shao**、**David Tarditi** 和 **Andrew Tolmach**。

一、总览

ML 是一种严格求值（strict）、高阶（higher-order）的函数式编程语言，具有静态检查的多态类型（statically checked polymorphic types）、垃圾回收（garbage collection）机制，并且拥有完整的形式化语义（formally defined semantics）。

Standard ML of New Jersey（SML/NJ）是 ML 语言的一个优化编译器和运行时系统。它在优化和代码生成过程中使用的一种中间表示——续体传递风格（*Continuation-Passing Style, CPS*），不仅适用于 ML 语言的编译，还广泛适用于许多现代编程语言的编译。本书的主题正是基于续延传递风格的编译技术。

事先了解 ML 语言对于阅读本书有所帮助，但并非必需。由于 *Standard ML of New Jersey* 编译器本身是用 ML 语言编写的，我们会在许多示例中使用 ML 语法。不过，我们仅使用该语言的一个简单子集进行说明，并在讲解过程中逐步解释相关符号。完全不熟悉 ML 的读者可以参考附录 A 中的介绍。

1.1. 续体传递风格（CPS）

FORTRAN 的优雅之处——也是它相较于汇编语言的一个重要进步——在于它让程序员无需为中间结果命名。例如，我们可以直接写：

$$x = (a + b) \times (c + d)$$

而不必使用汇编语言的形式：

$$\begin{aligned} r_1 &\leftarrow a + b \\ r_2 &\leftarrow c + d \\ x &\leftarrow r_1 \times r_2 \end{aligned}$$

这一简单的单行表达式比三行汇编代码更易读、更易理解，因为像 r_1 和 r_2 这样的中间变量名称并没有真正帮助我们理解计算过程。此外，汇编代码明确指定了计算的求值顺序：先计算 $a + b$，然后计算 $c + d$。但在多数情况下，这一细节并非我们真正需要关心的内容。

λ-演算（lambda calculus）在处理函数值时也提供了与 FORTRAN 类似的优势。我们可以直接写：

$$f(\lambda x.x + 1)$$

而不必像 Pascal 语言那样显式定义一个命名的函数：

```
function g(x: integer): integer;
    begin g := x+1 end;
    . . . f(g) . . .
```

在 Pascal 中，我们被迫为这个函数赋予一个名称（g），但这个名称（以及随之而来的冗长定义）未必能真正帮助我们理解程序的逻辑。

此外，λ-演算还能使我们摆脱对求值顺序的过度关注，即使在函数调用的边界之间也是如此。这种特性在简化程序表达的同时，也增强了灵活性。

这些便利性对于人类编写程序来说是非常合适的，但对于编译器处理程序时，可能恰恰相反。编译器通常希望为程序中的每个中间值赋予一个唯一的名称，以便利用这些名称进行表查找、值集合的操作、寄存器分配、指令调度等优化工作。

续体传递风格（*CPS*）是一种程序表示方式，它使得控制流和数据流的每个细节都变得显式可见。此外，CPS 还具有一个重要优势：它与 *Church* 丘奇λ-演算密切相关，而 λ-演算本身具有明确且公认的数学语义。

我们可以用一个示例来非正式地说明 *CPS*。先从以下源程序开始：

```
fun prodprimes(n) =
    if n = 1
    then 1
    else if isprime(n)
         then n * prodprimes(n - 1)
         else prodprimes(n - 1)
```

这是一个 ML 程序，它计算小于或等于正整数 n 的所有素数的乘积。关键字 fun 用于引入函数定义；等号右侧的表达式是函数体。if-then-else 语句和算术表达式的表示方式应当对大多数读者而言是熟悉的。

现在，这个程序的控制流中有几个值得命名的关键点。例如，当 isprime 函数被调用时，它会被传递一个返回地址，我们称其为 k。isprime 将返回一个布尔值，我们称其为 b。在 then 分支中的第一次 prodprimes 调用会返回到某个位置 j，并带有一个整数 p；而 else 分支中的第二次 prodprimes 调用会返回到位置 h，并带有整数 q。第一次计算 $n-1$ 的结果会存储在变量 m 中，而第二次计算的结果存入变量 i，依此类推。

我们还应提到，当 prodprimes 被调用时，它会被传递一个返回地址，我们可以称其为 c，并将其视为该函数的一个参数（形式参数）。然后，当函数需要返回时，我们可以使用 c 来继续执行后续操作。

我们可以使用续体来表达这一点。 续体（continuation） 是一个表示“下一步该做什么”的函数。例如，我们可以说 prodprimes 作为参数之一接收一个续体 c，并且当 prodprimes 计算出其结果 a 时，它将通过将 c 应用于 a 来继续执行。因此，函数返回的过程看起来就像是一次函数调用！

以下程序是上述程序的续体传递风格（*CPS*）版本，使用 ML 编写。对于不熟悉 ML 的读者，这里需要解释 let ... in ... end 结构：其中 let 声明函数、整数等局部值，其作用域包含 in 后的表达式，而 let 结构的最终结果就是该表达式的结果。fun 语句用于声明一个带有形式参数的函数，而 val 语句（在这个简单的示例中）用于将变量绑定到一个值。

```
fun prodprimes (n, c) =
  if n = 1
  then c(1)
  else let fun k (b) =
                if b = true
                then let fun j (p) =
                             let val a = n * p
                             in c(a) end
                         val m = n - 1
                     in prodprimes (m, j) end
                else let fun h(q) = c(q)
                         val i = n - 1
                     in prodprimes (i, h) end
       in isprime (n, k) end
```

可以注意到，之前讨论的所有控制点 c、k、j、h 都只是续体函数，而所有的数据标识符 b、p、a、m、q、i 仅仅是变量。为了使用续体传递风格（CPS），我们并不需要大幅修改原有的表示方式；我们只是使用了该语言的一种受限形式。关于 CPS 的完整解释将在第 2 章和第 3 章中给出。

CPS 表示易于优化编译器进行操作和转换。例如，我们希望执行尾递归消除：如果函数 f 在其执行的最后一步调用了函数 g，那么与其将 g 的返回地址设为 f 内部的某个地址，不如直接将 f 从其调用者那里接收到的返回地址传递给 g。这样，当 g 返回时，它会直接返回到 f 的调用者，而无需额外的中间跳转。

如果我们回顾 prodprimes 的原始版本，会发现其中有一个尾递归调用（即 prodprimes 的最后一次递归调用）。在 CPS 版本中，这种尾递归的特性表现为续体函数 h 是平凡的，即：

```
fun h(q) = c(q)
```

当我们有一个像 h 这样的函数，它只是简单地调用另一个函数 c 并传递相同的参数时，我们可以认为 h 等价于 c，因此可以直接用 c 替换 h。这样，我们可以进行如下转换：

转换前

`let fun h(q) = c(q)`
`  val i = n - 1`
`in prodprimes(i, h)`
`end`

转换后

`let val i = n - 1`
`in prodprimes(i, c)`
`end`

这样，我们就干净利落地完成了尾递归消除。

1.2. CPS 的优势

使用续体传递风格（CPS）作为编译和优化的框架有许多理由。在本讨论中，我们将其与几种替代方案进行比较：

- λ：λ-演算（不包含显式的续体）。对于像 ML 和 Scheme 这样“基于 λ-演算”的语言来说，λ-演算可能是一个合适的理论工具，用于推理其行为。
- QUAD：寄存器传输（或“四元式”），大致对应于极简冯·诺伊曼机器的指令集。
- PDG：程序依赖图（Program Dependence Graphs），能够在尽可能减少耦合的情况下同时表示控制流和数据流。
- SSA：静态单赋值形式（Static Single Assignment, SSA [2]，特别适用于高效实现某些数据流分析算法。在 SSA 中，每个变量仅被赋值一次；当控制流路径合并时，使用显式的传输函数来维持单赋值的假象。SSA 和 CPS 在某些重要方面是相似的，因为 CPS 也具有某种单赋值特性。

这些中间表示（IRs）的设计目标各不相同，以便支持不同类型的代码转换和优化。接下来，我们将考察几种不同类型的优化，并分析在每种表示方式下，它们的实现难度如何。

1.2.1. 内联展开（In-line Expansion）

λ-演算的变量绑定和作用域规则特别适用于 β-归约（β-reduction），即函数的内联展开：函数体被替换为函数调用的具体实现，并且实际参数替换形式参数。

然而，使用 λ-演算来表达严格求值（strict call-by-value）语言（如 ML、Pascal、Lisp、Scheme、Smalltalk 等）的行为存在一些问题。在这些编程语言中，函数参数应当在函数体执行之前求值，但在 λ-演算中并不需要遵守这一规则。

按照 λ-演算的 β-归约规则，实际参数会直接替换函数体中的形式参数，但这可能会导致以下问题：

- 一个本应进入无限循环的程序（在严格求值语义下）可能会终止。
- 在原始程序中本应仅被求值一次的实际参数，可能会被求值多次（如果函数体内多次使用该参数）。
- 在具有副作用的语言（如 ML、Pascal、Lisp 等）中，实际参数的副作用可能会：
  - 发生在函数体的部分副作用之后，
  - 完全不发生，
  - 发生多次。

在 CPS 中，执行参数替换同样很容易，但它不会遇到上述问题。所有函数的实际参数都是变量或常量，不会是复杂的子表达式。因此，实际参数替换形式参数（以及随之发生的参数“移动”到函数体内部）不会引发意外行为。

至于其他表示方式：

- QUAD、PDG、SSA 主要关注单个函数体的表示，而不太关注跨函数边界的优化。
- 在这些框架中，仍然可以实现内联展开，但需要额外机制来表示函数参数和调用序列。
- 但是，它们仍然需要解决终止性问题和副作用的乱序执行问题。

1.2.2. 闭包表示（Closure Representations）

在具有 块结构（block structure） 或 嵌套函数（nested functions） 的语言（如 Pascal、Scheme、ML）中，一个函数 f 可能嵌套在函数 g 内，而 g 又可能嵌套在另一个函数 h 内。在这种情况下，f 不仅可以访问自己的形式参数和局部变量，还可以访问 g 和 h 的形式参数和局部变量。

编译器的一个重要任务是高效地实现这种变量访问。

由于 λ-演算 和 CPS 都允许函数具有嵌套作用域，编译器可以更容易地操作这些函数及其表示形式，以计算非局部变量的高效访问方法。

相比之下，QUAD、PDG 和 SSA 主要关注单个函数体内的控制流和数据流，因此它们不易直接解决跨作用域的变量访问问题。

1.2.3. 数据流分析（Dataflow Analysis）

数据流分析涉及静态地传播值（更准确地说，是编译时标记，代表运行时的值）沿着控制流图进行计算。它能够回答诸如“变量的这个定义是否能到达那个使用点？”的问题，这对于某些优化非常有用。

由于续体传递风格（CPS）能够较为准确地表示控制流图，因此数据流分析在 CPS 中和在更传统的表示方式（如 QUAD）中一样容易进行。

静态单赋值形式（SSA）的设计使得前向数据流分析特别高效，因为它能轻松确定变量的任何使用点对应的唯一赋值点——即，每个变量仅被赋值一次。我们将在后续讨论中看到，CPS 具有类似于单赋值的特性。

另一方面，λ-演算 并不太适用于数据流分析。

1.2.4. 寄存器分配（Register Allocation）

在将程序中的变量分配到机器寄存器时，使用一种能够清晰表示变量的生命周期（创建和销毁）的表示方式是十分有用的。这种活跃性分析（liveness analysis）本质上是一种数据流分析，因此前面关于数据流分析的讨论同样适用于寄存器分配。

特别值得注意的是，在 CPS-为基础的编译器的某些阶段，CPS 表达式中的变量与目标机器的寄存器有着非常紧密的对应关系。

1.2.5. 向量化（Vectorizing）

程序依赖图（PDG） 尤其适用于优化，例如将普通循环转换为向量指令的优化。在其他中间表示（IRs）中，仍然可以实现类似的优化，但通常需要额外的数据结构来完成 PDG 所提供的功能。

1.2.6. 指令调度（Instruction Scheduling）

现代的深度流水线计算机要求编译器在后端阶段进行指令调度，以避免运行时的 流水线互锁（pipeline interlocks） 。指令调度涉及对单个指令的操作，并需要详细了解它们的大小、执行时间和资源需求。本章讨论的中间表示（IRs）可能过于抽象，并不直接适用于编译器的指令调度阶段。

1.2.7. 结论

本章介绍的中间表示（IRs） 具有许多相似之处：

- 静态单赋值形式（SSA） 只是 四元式（QUAD） 的一种受限形式。
- 续体传递风格（CPS） 只是 λ-演算 的一种受限形式。
- SSA 和 CPS 之间也存在许多相似性，因为 CPS 变量具有 单一绑定（single-binding） 属性。

通过 continuations 续体，我们既可以获得 λ-演算 的清晰替换操作，又可以实现适用于冯·诺伊曼机器的数据流分析和寄存器分析。

1.3. ML 语言简介

本书将展示如何在实际编译器中使用续体进行编译和优化。

我们的编译器——Standard ML of New Jersey（SML/NJ）——用于编译 ML 语言；但续体传递风格（CPS）并不局限于 ML，它已被应用于多种语言的编译器中[3]。

ML 编程语言最初于 1970 年代末被开发，作为 爱丁堡可计算函数逻辑（LCF） 定理证明系统的元语言（Meta-Language）[4]。 在 1980 年代初，人们逐渐认识到 ML 本身是一种有用的编程语言（即使对于不从事定理证明的人而言也是如此），并实现了一个独立的 ML 系统 [5]。 自那以来，Standard ML 语言 被正式定义 [6]，其形式语义也得到了书写 [7]，并且已经有多个编译器 可用 [8], [9]。目前，在众多机构和场所，已有数百名程序员积极使用 该语言。

ML 作为一种实用的编程语言，具有以下优势：

- ML 是严格 strict 求值的——即，函数的参数在函数调用之前被求值。这与 Pascal、C、Lisp 和 Scheme 相同，但不同于 Miranda 和 Haskell，后者是惰性 lazy 求值的。- ML 支持 高阶函数 higher-order functions，即一个函数可以作为参数传递，也可以作为另一个函数的返回值。这类似于 Scheme、C 和 Haskell，但不同于 Pascal 和 Lisp。然而，与 C 不同，ML 还支持 嵌套函数 nested functions（Scheme 和 Haskell 也支持），这使得高阶函数更加有用。- ML 具有 参数化多态 parametric polymorphic 类型，即一个函数可以应用于多个不同类型的参数，只要它对参数执行的操作与类型无关。Lisp、Scheme 和 Haskell 也支持参数化多态，而 Pascal 和 C 不支持。参数化多态不同于 重载 overloading，后者要求针对不同类型提供不同的函数实现。- ML 采用 静态类型检查 statically checked types，即类型检查在编译时完成，因此运行时不需要类型检查（这也能在程序运行前发现许多错误）。其他静态类型检查的语言包括 Pascal、C、Ada 和 Haskell；而 Lisp、Scheme 和 Smalltalk 则在运行时进行动态类型检查。然而，ML（与 Haskell 类似）具有 类型推导 type inference，程序员无需显式声明大多数类型。而在 Pascal、C、Ada 以及其他 Algol 系 语言中，程序员必须显式声明每个变量的类型。- ML 具有 垃圾回收 garbage collection，能够自动回收不可达的存储空间。这是 Scheme 和 Haskell 等函数式语言的典型特性，但 Lisp 和 Smalltalk 也支持垃圾回收，而 Pascal、C 和 Ada 通常不支持。- ML 采用静态作用域 lexical scoping——在程序文本中，变量的声明位置决定了其作用范围。这与 Pascal、C 和 Scheme 相同，但不同于 Smalltalk，后者对函数采用 动态绑定 dynamic method lookup。- ML 支持 副作用 side effects，包括输入/输出以及可变 引用变量 reference variables，可以执行赋值和更新操作。在这一点上，它与 Pascal、C、Smalltalk、Lisp 和 Scheme 等语言相同，但不同于 Haskell 这类纯函数式语言。不过，在 ML 中，可变变量和数据结构受到静态类型系统的约束，并且在编译时可静态识别。在典型的 ML 程序中，绝大多数变量和数据结构都是不可变的。- ML 具有完整的 形式语义 formally defined semantics [7]，保证了：- 所有合法程序的执行结果是 确定的 deterministic。
  - 所有非法程序都能被编译器识别为非法。 这与 Ada 形成对比——尽管 Ada 也有形式化语义，但它并不完整，因为某些“错误的”程序仍然必须被编译器接受。此外，Pascal 存在“歧义和不安全性” [10]，而 C 则因指针的不安全操作，很容易让程序员误修改运行时环境的任意部分。Lisp 和 Scheme 在这方面表现较好——原则上，“错误的”程序会在编译时或运行时检测到，但某些行为（如参数的求值顺序）仍未明确规定 [11]。

从上述总结可以看出，我们的编译器需要处理的问题以及 ML 语言的特性，与其他编程语言的编译器所面临的挑战有许多相似之处。然而，“现代”语言（如 ML） 的编译器必须在多个方面与“传统”语言（如 C）的编译器有所不同：

- 高阶函数 higher-order functions（如 ML、Scheme、Smalltalk 等）要求编译器在运行时引入数据结构来表示这些函数的自由变量。由于这些 闭包 closures 的生命周期通常无法在编译时确定，因此必须使用某种形式的 垃圾回收 garbage collection。- 垃圾回收的存在要求所有运行时数据结构 采用垃圾回收器可理解的格式；此外，编译后的代码所使用的机器寄存器 和 其他临

时变量 也必须对回收器可访问且可解析。- ML 鼓励“函数式”编程风格，即旧数据很少被更新，而是不断创建新数据。因此，垃圾回收器必须特别高效。在某些较早的 Lisp 系统，以及一些支持垃圾回收的 Algol 系语言 中，垃圾回收的负担较轻，因为新对象的分配频率较低。- 控制流主要通过源语言中的函数调用表示（而非 `while` 或 `repeat` 之类的内建控制结构）。因此，编译器必须确保函数调用（尤其是尾递归调用） 的开销极低。- ML 不提供“ 宏 ”功能。在 C 和 Lisp 中，宏通常用于对频繁执行的代码片段进行 内联展开 。对于不支持宏的语言，一个优秀的编译器 应当自动进行函数的内联展开，这是一种更安全的优化方式，并且可以提供类似的性能提升。

ML 具有一个独特的特性，这是其他常见编程语言所不具备的：大多数数据结构在创建后是不可变的。也就是说，一旦一个变量、列表单元或堆上的记录被创建并初始化，它就不能被修改。 当然，每次调用函数时，函数的局部变量都会被重新实例化，并赋予不同的值；堆上的列表单元最终会变成垃圾（因为没有局部变量再指向它们），而新的单元会不断创建。但由于列表单元是不可修改的，因此 别名 问题变得微不足道。 在传统编程语言的编译过程中，以下语句不能交换执行顺序：

$$a \leftarrow \#1(p);$$
$$\#1(q) \leftarrow b$$

（其中 $\#1(x)$ 表示 x 指向的记录的第一个字段），因为 p 和 q 可能是别名，即它们可能指向同一个对象。 同样，以下语句不能交换，除非对 f 的行为有足够的了解：

$$a \leftarrow \#1(p);$$
$$b \leftarrow f(x)$$

在 ML 中， 可变变量 和可变数据结构（即在创建后可以被修改的对象）具有与不可变对象不同的静态类型。换句话说，它们在编译器的类型检查阶段就已经被区分开来。在典型的 ML 程序中，绝大多数变量和数据结构都是不可变的。 因此， 别名 问题在 ML 中基本上消失了：如果 `p` 是一个不可变变量（通常如此），那么从 `p` 读取数据的操作几乎可以与任何其他操作交换顺序而不影响结果。我们的编译器在多个方面利用了这一特性。 惰性语言（如 Haskell）在原则上也具有不可变变量，但实际上，当一个变量被更新（即用求值后的结果替换原先的惰性求值单元）时，对于编译器和运行时系统的某些部分而言，这种操作看起来仍然类似于一次变量修改。 最后，ML 拥有高度抽象的 形式语义 ，这在许多方面都极为有用。任何不会改变可计算函数本质的优化或表示变换都是合法的。 相比之下，C 语言没有正式的形式语义。即使我们可以对 C 语言的行为形成一个合理的非正式理解，这种“语义”仍然高度依赖于底层机器的表示。此外，许多 C 程序违反了语言规范，但仍然被认为应该能在“合理的”编译器上运行。 在这样的情况下，C 编译器对程序转换的自由度非常有限。

1.4. 编译器组织结构（Compiler Organization）

Standard ML of New Jersey（SML/NJ）[8] 是一个 ML 语言的编译器，并且它本身是用 ML 语言编写的。它是一个 多遍 编译器，通过一系列阶段 将源程序转换为机器语言程序。其主要过程如下：

1. 词法分析 、解析 、类型检查，并生成带注释的抽象语法树。2. 转换为类似 λ-演算的中间表示（详见第 4 章）。3. 转换为续体传递风格（CPS）（详见第 5 章）。4. 优化 CPS 表达式，生成更优的 CPS 表达式（详见第 6–9 章）。5. 闭包转换 ，生成一个闭合的 CPS 表达式，即其中的每个函数都不含自由变量（详见第 10 章）。6. 消除嵌套作用域，生成仅包含一个全局 互递归 函数集的 CPS 表达式，这些函数都是非嵌套定义的（详见第 10 章）。7. 寄存器溢出处理，确保任何 CPS 表达式中的子表达式 至多拥有 n 个自由变量，其中 n 受目标机器的寄存器数量限制（详见第 11 章）。8. 生成目标机器的“汇编语言”指令，但此时仍为抽象形式（非文本形式）（详见第 13 章）。9. 指令调度 、 跳转优化 、 回填 ，最终生成目标机器指令（详见第 14 章）。

本书的结构与编译器的架构基本相同，但不包括第一阶段，即 ML 语言的前端处理。事实上，编译器的“前端”部分比“后端”要复杂得多，但本书的重点是如何使用 continuations 续体 进行优化和代码生成，而不是如何编译 ML。

二、CPS 形式

最早使用续体传递风格 (CPS) 作为中间语言 (intermediate language) 的编译器 [83, 54]，使用 Scheme [12] 语法来表示 CPS——在这些编译器中，CPS 表达式 只是一个满足特定语法约束的 Scheme 程序。在 Standard ML of New Jersey 编译器（以及本书）中，我们采用了一种更专门化的表示方式。在我们的实现中，CPS 表达式树 由 ML 数据类型 (datatype) 表示；这种 datatype 语法本身 能够自动保证大多数 Scheme 方法中仅作为约定的语法特性。此外，我们还做了如下改进：

- 每个函数都有一个名称，而不是匿名函数。
- 使用语法运算符来定义 互递归 (mutually recursive) 函数，而不仅仅依赖“外部”不动点 (fixed-point) 函数。- 支持n 元组 (n-tuple) 运算符，使得记录 (record) 和闭包 (closure) 的表示更加方便。

本章将介绍 CPS 作为程序中间表示（IR）的具体数据结构，并非正式地描述 CPS 的工作方式。 CPS 表达式的一个重要性质 是：函数（或原语操作符，如 +）的所有参数必须是 原子值 (atomic values) ——即变量或常量。一个函数调用不能作为另一个调用的参数。 这一设计的原因是，CPS 语言的目标是模拟冯·诺伊曼 (von Neumann) 机器的执行方式。在这种架构中，计算机一次只能执行一个操作，并且要求所有运算所需的参数事先已经准备好并存储在寄存器中。

2.1. CPS 数据结构

我们直接在 ML 数据类型 cexp（续体表达式）中表达这些限制，如 Listing 7 所示。

```
signature CPS =
sig
  eqtype var
  datatype value =
    VAR of var
  | LABEL of var
  | INT of int
  | REAL of string
  | STRING of string
  datatype accesspath = OFFp of int | SELp of int * accesspath
  datatype primop =
    * | + | - | div | ~ | ieql | ineq | < | <= | > | >=
  | rangechk | ! | subscript | ordof | := | unboxedassign
  | update | unboxedupdate | store | makeref | makerefunboxed
  | alength | slength | gethdlr | sethdlr | boxed
  | fadd | fsub | fdiv | fmul | feql | fneq | fge | fgt | fle | flt
  | rshift | lshift | orb | andb | xorb | notb
  datatype cexp =
    RECORD of (value * accesspath) list * var * cexp
  | SELECT of int * value * var * cexp
  | OFFSET of int * value * var * cexp
  | APP of value * value list
  | FIX of (var * var list * cexp) list * cexp
  | SWITCH of value * cexp list
  | PRIMOP of primop * value list * var list * cexp list
end
```

Listing 7: The CPS data type

对于不熟悉 ML 的读者，这里需要解释 datatype 关键字：它定义了一种“不相交联合 (disjoint-union)”或“变体记录 (variant-record)”类型（详见附录 A）。在 cexp 类型中，每个值都带有一个“标签”（构造器 (constructor)），例如 RECORD、SELECT、OFFSET 等。如果 cexp 的标签是 SELECT，那么它会包含四个字段，分别是 整数 (integer)、值 (value)、变量 (var) 和续体表达式 (cexp)，依此类推。 每个续体表达式 (continuation expression) 都会：- 接受 零个或多个 原子参数 (atomic arguments)，- 绑定 零个或多个结果 (results)，- 继续执行 零个或多个

continuation expressions
续体表达式 。例如，以下 ML 表达式表示整数加法：将变量 a 和 b 相加，得到结果 c，然后继续执行表达式 e：

```
PRIMOP(+, [VAR a, VAR b], [c], [e])
```

list datatype
（对于 ML 初学者：方括号 `[]` 表示列表，而 `VAR` 只是一个用户定义的 值类型 的构造器。因此，`PRIMOP` 的第二个参数是一个包含两个值的列表，第三个参数是一个包含一个变量的列表，依此类推。此外，`+` 在这里是一个 identifier
标识符，与 a 或 `PRIMOP` 相同；通常，它被绑定到一个加法函数，但在这里，它被重新绑定为一个不带参数的数据类型构造器。整个 ML 表达式 `PRIMOP(...)` 仅仅是构造了一个数据结构，用于表示编译器可能需要操作的 continuation expression
续体表达式 。）

atomic
在 CPS 中，操作的参数必须是原子值，即它们可以是变量或常量，但不能是复杂的子表达式。这正是 CPS
续体传递风格 的核心思想。例如，通常我们可以写出如下表达式：

```
e = (a + 1) * (3 + c)
```

但在 CPS 中，必须为所有子表达式创建名称，从而将其分解为：

```
u = a + 1
v = 3 + c
e = u * v
```

使用 CPS 数据类型 来表示这一计算过程，则可以写成：

```
PRIMOP( +, [VAR a, INT 1], [u],
  [PRIMOP( +, [INT 3, VAR c], [v],
    [PRIMOP( *, [VAR u, VAR v], [e], [M])])])
```

其中，我们假设续体表达式 M 在后续计算中使用了变量 e。

operators atomic arguments union
数据类型 `value` 是 CPS 操作符 可以接受的所有 原子参数 的 联合。每个参数可以是：- 变量（VAR）- 整数常量（INT）- 字符串常量（STRING）- 浮点数常量（REAL） 关于 `LABEL` 的用法将在 Section 2.4 进行解释。

虽然数学表达式 $(a+1)\times(3+c)$ 不指定 $a+1$ 和 $3+c$ 哪个应先求值，但在转换为 续体传递风格（CPS） 时，必须在二者之间做出选择。在前面的示例中，$a+1$ 被设定为先求值。这也是 CPS 的一个核心特性：许多控制流的决策在源语言转换为 CPS 时已经确定。然而，这些决策并非不可逆。经过适当的分析，优化器仍然可以调整 continuation expression
续体表达式 ，使得 $3+c$ 先求值。

continuations
续体 可以非常自然地表达控制流。整数比较运算符（`>`）接受两个整数参数（常量或变量），不返回结果，并在比较后选择两个续体表达式之一继续执行。例如，条件表达式 `if a > b then F else G` 在 CPS 中可表示为：

```
PRIMOP(>, [VAR a, VAR b], [ ], [F, G])
```

multiway branches indexed jumps
其中 F 和 G 是续体表达式。 多路分支 （即 索引跳转 ）可以通过 `SWITCH` 运算符表示，例如：

```
SWITCH(VAR i, [E0, E1, E3, E4])
```

该表达式根据 i 的值（0、1、3、4）决定继续执行 E_0、E_1、E_3 或 E_4。如果 i 的运行时值小于 0 或超出列表索引范围，则会导致错误的求值行为。续体表达式还可以用于在堆上构造 n-tuples
n 元组，并访问记录的字段。例如：

```
RECORD([(VAR a, OFFp 0), (INT 2, OFFp 0), (VAR c, OFFp 0)], w, E)
```

该表达式在堆上创建一个包含 3 个字段的记录，初始化为 $(a,2,c)$，并将其地址绑定到变量 w，然后继续执行 E。其中 `OFFp 0` 的含义将在后续介绍。需要注意的是，所有 `RECORD` 创建的堆对象都是 immutable
不可变的，即它们不能被修改或重新赋值。

表达式 `SELECT(i, v, w, E)` 取出记录 v 的第 i 个字段，并将结果绑定到 w，然后继续执行 E。如果 i 所指定的字段不存在，则该表达式没有意义。字段的编号从 0 开始。在 CPS 语言中，变量可以指向一个记录值，但它也可

能指向记录的中间某个字段。`OFFSET` 原语允许调整指针；如果变量 v 指向记录的第 j 个字段（j 可能等于 0），那么 `OFFSET(i, v, w, E)` 使 w 指向记录的第 $(j+i)$ 个字段，并继续执行 E。常量 i 可以是负数，但 $j+i$ 必须为非负数。

互递归（mutually recursive）函数使用 `FIX` 运算符定义。表达式

```
FIX([(f1, [v11, v12, ..., v1m1], B1),
     (f2, [v21, v22, ..., v2m2], B2),
     ...
     (fk, [vk1, vk2, ..., vkmk], Bk)],
    E)
```

定义了零个或多个函数 $f_1, ..., f_k$，这些函数可以在表达式 E 中被调用，也可以在彼此的函数体 $B_1, ..., B_k$ 内相互调用。每个函数 $f_1, ..., f_k$ 的形式参数由变量 v_{ij} 组成。执行 $\text{FIX}\left(\vec{f}, E\right)$ 的效果是定义列表 $\vec{f}$ 中的所有函数，然后执行 E。通常，E 通过 APP 运算符调用一个或多个 f_i，或者将 f_i 作为参数传递给另一个函数，或者将某些 f_i 存储在数据结构中。

在 续体传递风格 (CPS) 中，所有函数调用都是尾调用 (tail calls)，也就是说，它们不会返回到调用它的函数。这一点可以从 APP 续体表达式 (continuation expression) 的形式看出，因为它不包含任何续体表达式作为子表达式。

例如，表达式 `SELECT(3, v, w, SELECT(2, w, z, E))` 依次执行以下操作：

1. 取出 v 的第三个字段，存入 w，
2. 继续取出 w 的第二个字段，存入 z，
3. 然后继续执行 E。

然而，APP 表达式 `APP(f, [a1, a2, ..., ak])` 仅执行函数调用 `f(a1, a2, ..., ak)`，并不执行其他任何操作。也就是说，函数 f 的函数体 直接在新的求值环境中执行，实际参数 a_i 被替换为 f 的形式参数。

函数 f 具有由 `FIX` 声明绑定的形式参数和函数体。当然，由于函数是“第一类值 (first-class)”，它们可以作为参数传递或存储到数据结构中，因此变量 f 可能实际上是某个 静态封闭 (statically enclosing) 函数的形式参数，或者是 `SELECT` 操作的结果等。然而，无论 f 以何种方式出现，它所引用的值最初必须由 `FIX` 运算符创建。

当然，在大多数编程语言中，函数是允许返回到调用者的！但是，回想 Section 1.1) 中的示例，其中 `prodprimes` 的某次调用可以直接返回到其调用者的调用者。这是因为，无论 `prodprimes(n-1)` 返回什么值，该值都会直接作为 `prodprimes(n)` 的返回结果。如果所有函数调用都是这样的，那么直到程序执行结束，所有函数才会最终返回。在这种情况下，就不再需要运行时维护返回地址（以及局部变量等）的 栈 (stack) 来记录“应该返回到哪里”。

在 续体传递风格 (CPS) 编译器中，所有函数调用都必须被转换为尾调用 (tail calls)。这通过引入 续体函数 (continuation functions) 来实现：续体表达的是被调用函数本应返回后，剩余的计算过程。续体函数使用普通的 `FIX` 运算符进行绑定，并作为参数传递给其他函数。引入续体函数的具体算法将在 Section 5 详细讲解。

一个简单的示例可以说明这一点。假设我们有如下源语言程序：

```
let fun f(x) = 2*x+1
 in f(a+b)*f(c+d)
end
```

该程序可以（大部分）转换为续体传递风格 (CPS)，如下所示：

```
let fun f(x, k) = k(2x+1)
    fun k1(i) = let fun k2(j) = r(ij)
                 in f(c+d, k2)
                end
 in f(a+b, k1)
end
```

其中，r 代表计算的 “剩余部分” (rest of the computation)，即原始程序片段本应返回其计算结果的地方。在这个转换后的程序中，每个函数调用都是封闭函数执行的最后一步，因此它可以直接被转换为 CPS 语言：

```
FIX([(f, [x, k], PRIMOP( *, [INT 2, VAR x], [u], [
                  PRIMOP(+, [VAR u, INT 1], [v], [
                  APP (VAR k, [VAR v])])])),
     (k1, [i], FIX([(k2, [j], PRIMOP( *, [VAR i, VAR j], [w], [
                               APP (VAR r, [VAR w])]))],
                PRIMOP (+, [VAR c, VAR d], [m], [
                APP (VAR f, [VAR m, VAR k2])]))))],
PRIMOP (+, [VAR a, VAR b], [n], [
APP (VAR f, [VAR n, VAR k1])]))
```

在这个转换中，k_1 和 k_2 被称为 continuations（续体），它们分别表示 f 的两次调用后的剩余计算。函数 f 被重写，使得它不再以普通方式返回，而是调用其续体参数 k，从而执行“剩余计算”。这种执行方式类似于普通编程语言中函数返回后继续执行后续代码的效果。在普通函数调用中，返回值通常会被保存并用于后续计算。而在 CPS 中，返回值仅仅是传递给续体函数的参数，如上例中 k_1 和 k_2 中的变量 i 和 j，它们分别捕获了 f 计算后的结果，并继续执行后续计算。

2.2. 逃逸的函数

考虑以下两个程序：

```
let fun f(a, b, c) = a + c
in f(1, 2, 3)
end
```

```
let fun g(a, b, c) = a + c
in (g, g(1, 2, 3))
end
```

函数 f 仅在局部使用，而 g 既在局部使用，也被导出到外部；我们称 g escapes（逃逸），而 f 没有逃逸。在 f 的情况下，我们可以执行一个简单的程序变换，去除未使用的参数 b，前提是我们同时修改所有对 f 的调用位置：

```
let fun f(a, c) = a + c
in f(1, 3)
end
```

但对于 g，这种变换就不那么容易了，因为我们无法确定所有 g 的调用位置。由于 g 可能被存储在某个数据结构中，并在程序的任意位置被取出调用；更糟糕的是，它甚至可能被导出并用于某个完全未知的“编译单元”，因此，我们不能改变它的表示方式——它必须仍然被看作一个接受三个参数（而非两个参数）的函数。

在使用 CPS 编译某种特定的编程语言时，我们可能需要对 escaping functions（逃逸函数）的行为施加一定的限制。这里我们以 ML 为例，其他编程语言可能会有完全不同的限制。对于不逃逸的函数（如上例中的 f），不需要任何限制，因为在任何可能需要分析其行为的调用点，函数体都可以通过语法轻松找到。在 ML 中，所有函数恰好接受一个参数。若需要多个参数，可以通过传递 n 元组 n-tuple 来实现；由于 ML 允许对 n 元组进行模式匹配，这种方式在语法上非常方便。在上例中，函数 f 和 g 实际上都是接受 3 元组作为参数的单参数函数。当 ML 代码转换为 CPS 时，每个用户定义的单参数函数都会变成一个双参数函数：新增的参数是 continuation function（续体函数）。而这些续体函数本身仍然是单参数函数。

因此，在 ML 转换为 CPS 时，我们对 escaping functions（逃逸函数）施加以下规则：

- 每个逃逸函数要么有一个参数，要么有两个参数。
- 如果一个逃逸函数有两个参数（即用户定义的函数），那么它的第二个参数始终是一个单参数的逃逸函数（即续体函数）。
- 当前的异常处理程序 exception handler[1]（通过 `sethdlr` 原语操作 `primop` 设定）始终是一个单参数的逃逸函数，即续体函数。

[1] ML 语言的 exception mechanism（异常机制）将会在本书中偶尔提及，并讨论其转换和优化。对于不熟悉 ML 异常的读者，可以忽略这些内容，因为它们并非本书的核心主题。

由于 ML 代码在转换为 CPS 时会自然地建立这些不变量（invariants），因此我们可能希望优化器（optimizer）能够利用这些不变量来推理逃逸函数的行为。由于逃逸函数可能来自不同的编译单元，我们无法直接通过检查其函数体推导这些信息，但可以确保优化器在所有编译单元中保持这些不变量。

对于允许多个参数的编程语言（如大多数命令式语言），我们可以制定类似的约定；无论如何，这些语言中都会存在用户定义的函数和续体函数。另一方面，在 Prolog 这样的逻辑编程语言中，每个谓词（predicate）可能需要两个续体（continuations），因此我们需要采用完全不同的约定。

2.3. 作用域规则

CPS 运算符生成的结果会绑定到词法作用域（lexical scope）内的变量。在同一个续体表达式（continuation expression）内，任何变量都不能被多次绑定，并且变量不能在其语法作用域（syntactic scope）之外被引用。CPS 的作用域规则简单且直接：

- 在表达式 `PRIMOP(p, vl, [w], [e1, e2, ...])` 中，变量 w 的作用域为 $e_1, e_2, \ldots$。
- 在表达式 `RECORD(vl, w, e)` 中，变量 w 的作用域仅限于 e。
- 在 `SELECT(i, v, w, e)` 或 `OFFSET(i, v, w, e)` 中，变量 w 的作用域仅限于 e。
- 在 `FIX([(v, [w1, w2, ...], b)], e)` 中，变量 w_i 的作用域仅限于 b，而 v 的作用域包含 b 和 e。这一规则可以推广到互递归函数（mutually recursive functions）的定义。例如，在 Listing 15 中每个 f_i 的作用域包含所有 B_j 及 E，而 v_{ij} 的作用域仅限于 B_i。
- `APP`、`SWITCH` 以及某些 `PRIMOP`（当第三个参数为空时）不绑定变量，因此不需要作用域规则。

每个变量的使用（use）（即作为 `value` 类型的一部分）必须位于其绑定（binding）的作用域内。一旦变量被绑定，它在其整个作用域内都保持相同的值，不能被重新赋值。当然，CPS 代码片段可能会被多次执行（例如，当它是一个被多次调用的函数体时）；在这种情况下，每次执行变量绑定时，可能会绑定不同的值。

2.4. 闭包转换

使用 CPS 作为编译器的中间表示（IR）的主要原因在于，它与冯·诺伊曼（von Neumann）计算机的指令集非常接近。大多数 CPS 运算符（如 `PRIMOP(+,...)`）都与机器指令高度对应。

然而，在冯·诺伊曼计算机上，函数定义的概念比 CPS 更原始。CPS 函数（类似于 λ-演算（λ-calculus）或 Pascal 的函数）可以包含自由变量（free variables），即表达式可以引用定义在最内层封闭函数之外的变量。例如，在第 15 页的示例中，函数 k_2 引用了 k_2 的形式参数 j，同时也引用了 i，而 i 是定义在 k_2 之外的变量。我们称 j 为 k_2 的绑定变量（bound variable），i 为 k_2 的自由变量（free variable）。当然，每个变量最终都会在某处被绑定，例如 i 是 k_1 的绑定变量。

冯·诺伊曼计算机仅通过机器代码地址来表示函数。这样的地址本身无法描述函数的自由变量的当前值。（至于函数的绑定变量，则不需要提前存储，因为这些变量只有在函数调用时才会获取值。）

通常，表示包含自由变量的函数的方法是闭包（closure）[13]，即一个包含机器代码指针和自由变量信息的二元组（pair）。例如，在 k_1 被调用并传递 `i = 7` 作为参数时，我们可以将 k_2 表示为记录（record）`(L2, 7)`；其中 L_2 是 k_2 的机器代码地址，而 7 是 k_2 这一实例中的 i 的值。

我们在 CPS 语言中显式表示闭包记录（closure records）。将 CPS 程序转换为另一个没有自由变量（free variables）的 CPS 程序，这一过程被称为“闭包转换（closure conversion）”。在闭包转换后，每个函数都会获得一个额外的参数，即闭包记录（closure record）。闭包记录的第一个字段是指向该函数本身的指针，其其余字段存储该函数的自由变量的值。

当一个“函数”被作为参数传递给另一个函数，或被存储到数据结构中时，实际上是闭包（closure）被传递或存储。而当另一个函数需要调用该闭包时，执行以下步骤：

1. 调用闭包 f 时，首先从 f 的第一个字段提取函数指针 f'。
2. 然后调用 f'，并将 f 作为第一个参数传递（其余参数依次传递）。

一个重要的特性是：f 的调用者无需知道 f 的闭包记录的具体格式，甚至不需要知道它的大小。调用者唯一需要了解的是如何提取函数的机器代码指针。而 f'（即 f 的实际函数体）知道如何在闭包记录中找到所需的自由变量。

我们使用 Listing 18 的示例（包含 f、k_1 和 k_2）来说明闭包转换，使用 ML 语法代替 CPS 语法（为了简洁）：

```
let fun f'(f'', x, k) = let val k' = #1(k)
                         in k'(k,2x+1)
                        end
    val f = (f')
    fun k1'(k1'', i) = let fun k2'(k2'', j) =
                               let val r' = #1(r)
                                in r'(r,#2(k2'')+j)
                               end
                           val k2 = (k2', i)
                       in f'(f, #2(k1'') + #3(k1''), k2)
                       end
    val k1 = (k1',c,d,f)
 in f'(f, a+b, k1)
end
```

（此代码在 ML 中可能无法正确类型检查，但它的目的仅是示意。）

函数 f 没有自由变量，因此它的闭包（closure）是最简单的：一个仅包含函数代码的单元素记录。但 k_1 有三个自由变量 c、d 和 f，因此它的闭包是一个四元素记录，包含：

- 这三个自由变量，
- k_1'（k_1 的转换版本），它被修改为期望一个闭包作为第一个参数。

一个关键点是：在每个闭包中，实际的函数始终是记录的第一个字段。这意味着，在不同上下文中调用闭包时，不需要知道其具体格式，只需提取第一个字段即可调用它。例如，在对 r 的调用中，尽管 r 的闭包可能包含多个字段，但调用它只需要提取第一个字段，并将整个闭包作为参数传递。在代码中：

- f' 接收它的闭包作为参数 f''，
- k_1' 接收它的闭包作为参数 k_1''，
- k_2' 接收它的闭包作为参数 k_2''。

另外，为了简化代码，我们在两处直接引用 f'，而不是先从 f 提取它。

Closure Conversion 阶段的输入是一个 CPS 表达式，它遵循上一节描述的作用域规则（scope rules）。转换的输出仍然是一个 CPS 表达式，但额外满足一个新的规则：所有函数都不再包含自由变量（free variables）。

更具体地说，在函数 g 的函数体中，唯一允许的自由变量是：

- 函数 g 的形式参数（formal parameters）。
- 函数名——即，在 FIX 表达式中，作为 `(function, formals, body)` 三元组的第一个元素出现的那些函数名。

在冯·诺伊曼（von Neumann）机器的实现中，函数名（即函数的机器代码地址）实际上是常量。因此，它们可以作为自由变量出现在其他函数中，而不需要通过闭包（closure）来查找。在闭包转换（closure conversion）的输出中，我们在 CPS 记法中使用 LABEL 而不是 VAR 来引用这样的变量，以表明它们本质上是常量，并且不再被视为自由变量。

因此，闭包转换后的 CPS 代码必须遵循以下自由变量规则（free-variable rule）：在一个互递归（mutually recursive）函数定义中，表达式：

```
FIX([
(f1, [v11, v12, ..., v1m1], B1),
(f2, [v21, v22, ..., v2m2], B2),
…
(fn, [vn1, vn2, ..., vnmn], Bn)],
E)
```

在 B_i 的自由变量 中，只能包含：

1. 函数 f_i 的形式参数 v_{ik}。
2. 所有 f_j（即 FIX 绑定的所有函数名）。

此外，所有对 f_j 的引用（以及仅有的 FIX 绑定的函数变量引用），在 CPS 记法 中必须使用 LABEL 构造器，而不是 VAR。

使用 LABEL 的主要原因是简化寄存器分配（register allocation）时的自由变量（free-variable）计算。从代码生成器（code generator）的角度来看：VAR 代表占用寄存器的变量，LABEL 代表不占用寄存器的常量。在 闭包转换（closure conversion） 完成后，所有函数都不再具有非平凡的自由变量，因此不再需要嵌套定义函数。我们可以将所有函数定义在顶层 FIX 作用域 中，使得整个 编译单元（compilation unit） 的形式变为：

```
FIX([
(f1, [v11, v12, ..., v1m1], B1),
(f2, [v21, v22, ..., v2m2], B2),
…
(fn, [vn1, vn2, ..., vnmn], Bn)],
E)
```

其中，B_i 和 E 都不再包含 FIX 操作符。

为了进一步简化，我们可以将 E 的自由变量转换为 E 的形式参数，从而得到：[2]

```
FIX([
(f0, [v01, v02, ..., v0m0], E),
(f1, [v11, v12, ..., v1m1], B1),
(f2, [v21, v22, ..., v2m2], B2),
. . .
(fn, [vn1, vn2, ..., vnmn], Bn)],
APP(VAR f0, [VAR v01, VAR v02, ..., VAR v0m0])])
```

编译器的后续阶段只需要处理形如 $(f_i, \vec{v}_i, B_i)$ 的三元组集合，而无需关注最终的 APP 表达式，因为它是平凡的（trivial），且不包含任何需要进一步分析的内容。

完整的闭包转换（closure conversion）算法将在 Section 10 详细描述。然而，在此之前，我们有必要正式定义表达式的自由变量（free variables）。我们引入辅助函数 fvl（free-variable list），它计算值列表中的变量集合：

$$
\begin{aligned}
\text{fvl}(\texttt{nil}) &= \emptyset \\
\text{fvl}((\texttt{VAR}\ v) :: l) &= \{v\} \cup \text{fvl}(l) \\
\text{fvl}((\texttt{LABEL}\ v) :: l) &= \text{fvl}(l) \\
\text{fvl}((\texttt{INT}\ i) :: l) &= \text{fvl}(l) \\
\text{fvl}((\texttt{REAL}\ i) :: l) &= \text{fvl}(l) \\
\text{fvl}((\texttt{STRING}\ i) :: l) &= \text{fvl}(l)
\end{aligned}
$$

基于 fvl，我们可以直接计算表达式的自由变量，这一过程非常直观：

[2] 译者注：在这个变换后，f_0 作为新的主入口函数，其形式参数 $v_{01}, v_{02}, \ldots, v_{0m_0}$ 是原本 E 的自由变量，它们被显式传递，而不再依赖外部作用域。

$$\mathrm{fv}(\texttt{APP}(v, l_a)) = \mathrm{fvl}(v :: l_a)$$
$$\mathrm{fv}(\texttt{SWITCH}(v, [C_1, C_2, ...])) = \mathrm{fvl}[v] \cup \bigcup_i \mathrm{fv}(C_i)$$
$$\mathrm{fv}(\texttt{RECORD}([(v_1, p_1), (v_2, p_2), ...], w, E)) = \mathrm{fvl}[v_1, v_2, ...] \cup \mathrm{fv}(E) - \{w\}$$
$$\mathrm{fv}(\texttt{SELECT}(i, v, w, E)) = \mathrm{fvl}[v] \cup \mathrm{fv}(E) - \{w\}$$
$$\mathrm{fv}(\texttt{OFFSET}(i, v, w, E)) = \mathrm{fvl}[v] \cup \mathrm{fv}(E) - \{w\}$$
$$\mathrm{fv}(\texttt{PRIMOP}(p, l_a, [w_1, ...], [C_1, ...])) = \mathrm{fvl}(l_a) \cup \bigcup_i \mathrm{fv}(C_i) - \bigcup_j \{w_j\}$$
$$\mathrm{fv}\Big(\texttt{FIX}\Big(\Big[\Big(f_1, \big[w_{11}, ..., w_{1m_1}\big], B_1\Big), ..., \Big(f_n, \big[w_{n1}, ..., w_{nm_1}\big], B_n\Big)\Big], E\Big)\Big) = \left(\mathrm{fv}(E) \cup \bigcup_i \left(\mathrm{fv}(B_i) - \bigcup_{j=1}^{m_i} \{w_{ij}\}\right)\right) - \bigcup_i \{f_i\}$$

2.5. 寄存器溢出（Spilling）

CPS 语言中的变量使用方式 在许多方面类似于冯·诺伊曼（von Neumann）机器上的寄存器（registers）。

- 算术运算符以变量（寄存器）作为操作数，并将结果存入变量（寄存器）。
- SELECT（内存读取）以变量（寄存器）和常量偏移量（offset）作为输入，并将结果存入变量（寄存器）。
- 其他操作也遵循类似的模式。

然而，冯·诺伊曼机器的寄存器数量是固定的，而 CPS 表达式可以拥有任意数量的变量。为了弥合这一差距，我们需要将多个 CPS 变量映射到相同的寄存器。但有一个限制：只有在两个 CPS 变量不会同时“存活（live）”时，它们才能共享同一个寄存器：即，在程序的后续计算中，只需要其中一个变量，而另一个变量已经不再被使用的时候。

在传统的数据流分析（dataflow analysis）中，一个变量在静态分析时被视为“活跃变量（live variable）”的条件是：该变量在某个点之后仍然被使用，但在该点之前没有重新绑定（reassigned）。

在 CPS 语言中，这一概念等价于 续体表达式（continuation expression）中的自由变量（free variables）。自由变量就是活跃变量！这一等价关系的证明很简单：只需注意，前面定义的自由变量函数（fvl）的计算方式，与传统数据流分析中的变量活跃性算法（针对 有向无环图（directed acyclic graph）的计算）完全相同。

对于 CPS，有 **有限寄存器规则（Finite-Register Rule）** 如下：

在 CPS 语言中，针对具有 k 个寄存器的目标机器，编译必须满足以下规则：

> CPS 代码中的任何子表达式的自由变量数量不得超过 k。

在 闭包转换（closure conversion）阶段之后，寄存器溢出（spilling）阶段会重写 CPS 代码，以确保满足该规则。

此外，某些编译阶段也受到有限寄存器规则的变体约束。例如，在溢出阶段（spill phase）之前，代码必须满足：

> 针对具有 k 个寄存器的目标机器，
> CPS 代码中的任何函数的 形式参数（formal parameters）数量不得超过 k。

完整的溢出阶段（spill phase）描述将在 Section 11 详细讲解。

三、CPS 的语义

CPS 表达式的含义可以通过一种简单的 denotaional semantics 指称语义 给出。完整的语义定义见 Section B；这里我们沿用该语义定义的 structure 结构，进行更为非正式的讨论。如果读者对 denotational syntax 指称语义 感到不适，可以只阅读本章正文，而略过代码部分的“语义”内容。所有 CPS 的变体——无论应用了作用域规则、自由变量规则以及语言相关规则的哪种子集——均遵循该语义。

下面给出的是一种直接的、使用 Standard ML 编写的 continuation semantics 续体语义。读者如果熟悉续体语义（参见例如 Stoy [14]、Gordon [15] 或 Schmidt [16]）以及 ML 语言（参见例如 Milner [7]、Reade [12] 或 Paulson [17]），将更易于理解下文。通过给出针对我们 CPS 表示的 formal semantics 形式化语义，我们希望能让读者独立验证后续章节所介绍的变换，尽管我们很少会给出正式的证明。

```
functor CPSsemantics(structure CPS: CPS ...
```

该语义以 Standard ML 中的 functor 函子形式编写。它以（形式上的）一个 CPS 结构（参见 Listing 7）作为参数，同时还包括：

```
val minint: int
val maxint: int
val minreal: real
val maxreal: real
```

我们不认为 CPS 作为模型所针对的底层机器架构应该具有无限精度的算术运算。必须存在某个最大和最小的可表示整数和实数。

```
val string2real: string -> real
```

CPS 语义将实数（浮点数）表示为字符串字面量，就像它们在源程序中被书写的方式一样（例如，`0.0`）。我们假设存在某种方法可以将其转换为机器表示。在编译的这一阶段，以字符串形式表示实数的实际原因是使 CPS 语言独立于特定的机器表示，从而使跨平台编译更加精确和容易。然而，这种表示方式的一个缺点是，使得实值表达式的 constant folding 常量折叠 变得非常困难。

在 denotational semantics 指称语义 中，具有对内存副作用的操作通常通过“存储”来表示。每个存储值可以被视为从 locations 位置（地址）到 denotable values 可指称值 的映射。位置的类型 `loc`，以及用于生成新位置的函数，都是语义的参数。该类型必须支持值的相等性测试（即为等 eqtype 价类型）。

```
val arbitrarily: 'a * 'a -> 'a
```

有些事物无法由语义预测。为了对这种不可预测性建模，表达式 `arbitrarily(a, b)`的求值结果可以是 `a` 或 `b` 之一。该表达式仅用于指针比较；关于相等性测试，详见 Section 4.3。

```
type answer
```

按照 continuation semantics 续体语义 的传统，我们引入一个类型 `answer`，用于表示程序整个执行过程的结果。我们实际上并不需要了解该类型的具体结构。

```
datatype dvalue =
 RECORD of dvalue list * int
| INT of int
| REAL of real
| FUNC of dvalue list -> (loc * (loc -> dvalue) * (loc -> int)) -> answer
| STRING of string
| BYTEARRAY of loc list
| ARRAY of loc list
| UARRAY of loc list
```

四、ML 特定的优化

五、CPS 变换

六、基于 CPS 的优化

6.1. 常量折叠与 β 规约

6.2. η 规约与柯里化

6.3. 级联优化

6.4. 实现

七、β 展开

7.1. 关于内联的时机

7.2. 估算优化效果

7.3. Runaway expansion

八、提升优化

8.1. 合并不动点定义

8.2. 提升优化的规则

8.3. 提升优化

十、闭包转换

十一、寄存器溢出

11.1. 表达式重排

11.2. 寄存器溢出算法

十二、空间复杂度

12.1. 空间分析的公设

12.2. 保持空间复杂度

12.3. 闭包内存布局

12.4. 关于启动垃圾回收的时机

十三、抽象机器

13.1. 编译单元

13.2. 与垃圾回收器的接口

13.3. 位置无关代码

13.4. 特殊目的寄存器

13.5. 伪指令

13.6. 续体机器的指令集

13.7. 寄存器分配

13.8. 分支预测

13.9. 抽象机器指令生成

13.10. 整数算术

13.11. 非装箱浮点值

十四、机器码生成

14.1. 翻译到 VAX

14.1.1. Span-dependent instructions

14.2. 翻译到 MC68020

14.3. 翻译到 MIPS 与 SPARC

14.3.1. PC-相关寻址

14.3.2. 指令调度

14.3.3. Anti-aliasing

14.3.4. Alternating temporaries

14.4. 一个例子

十五、性能评估

15.1. 硬件

15.2. 对每个优化步骤的测量

15.3. 调参

15.4. 关于缓存

15.5. 编译时

15.6. 与其他编译器的比较

15.7. 结论

十六、运行时系统

16.1. 垃圾回收的效率

16.2. 广度优先复制

16.3. 代际垃圾回收

16.4. 运行时数据格式

16.5. 关于分页机制

16.6. 异步中断

十七、并行

十八、未来方向

18.1. 控制流依赖性

18.2. 类型信息

18.3. 循环优化

18.4. 垃圾回收

18.5. 静态单赋值形式

18.6. 有状态多线程

本附录描述了 ML 语言的基础知识，足以理解本书其余部分中的示例。我们在此仅涵盖核心语言（不包含模块系统），但模块系统的概要参见第 4.8 节。如需更详细的内容，请参阅 Reade [68]、Paulson [67] 或 Sokolowski [81]。

Standard ML 的一个显著特点是其数据结构的表达能力。与其他语言类似，它包含原子类型（atomic types）（如整数）、笛卡尔积类型（cartesian product types）、记录（records）和不相交和类型（disjoint sum types）（类似于 Pascal 中的变体类型（variants）或 C 语言中的联合类型（unions）），但在 ML 中，这些概念的结合方式似乎更加安全和优雅。

原子类型（atomic types）包括 `int`（整数）、`real`（浮点数）和 `string`（零个或多个字符组成的序列）。

给定类型 $t_1, t_2, ..., t_n$（其中 $n \geq 2$），可以构造出笛卡尔积类型（cartesian product type）：$t_1 \times t_2 \times ... \times t_n$。（在 ML 的语法中，× 使用星号 * 表示。）这种类型称为 n-元组（n-tuple）；例如，`int*int*string` 就是一个三元组（3-tuple）类型，包含类似 `(3,6,"abc")` 这样的值。

记录类型（record type）是一种语法糖（syntactic sugaring），它在 n-元组（n-tuple）（其中 $n \geq 0$）的基础上，为每个「字段」（field）赋予了名称。例如，类型：

```
{wheels: int, passengers: int, owner: string}
```

包含的值形如：

```
{wheels=4, passengers=6, owner="Fred"}
```

虽然它与类型 `int*int*string` 非常类似，但两者并不能相互替代。

联合类型（union type）可以拥有几种不同形式的值；例如，一个列表（list）的值可以是一个对（pair），也可以是空值（nil）。与 C 语言中的联合（union）类型（或 Pascal 中的变体记录（variant record））不同，ML 要求每个值都必须带有一个标记（tag）以区分该值属于哪一种形式。在未首先检查标记（tag）之前，不可能从值中提取信息，因此 ML 中的联合类型更为安全。（ML 中将这种标记（tag）称为构造子（constructor）。）

使用关键字 `datatype` 来声明联合类型（union types）。在一个 `datatype` 声明中，有一个构造子（constructor）名称的列表，每个构造子（constructor）都指定了与之相关联的值的类型。例如：

```
datatype vehicle =
    CAR of {wheels: int, passengers: int, owner: string}
  | TRUCK of real
  | MOTORCYCLE
```

类型为 `vehicle` 的每个值都可以是汽车（car）、卡车（truck）或摩托车（motorcycle）。

如果是汽车（car），则携带一个记录（record）类型的值（两个整数和一个字符串）；如果是卡车（truck），则携带一个实数值（`real`），表示其总重量；而所有摩托车（motorcycle）的值都是相同的：`MOTORCYCLE` 称为常量构造子（constant constructor），因为它不携带任何值。

类型可以是多态的（polymorphic）：一个函数或构造子（constructor）可以操作具有不同（但类似）类型的对象。例如，`list` 数据类型可以用来构造整数列表、实数列表、整数列表的列表等等。

```
datatype 'a list = nil | :: of 'a * 'a list
```

符号 `::` 是一个标识符（identifier），就像 `TRUCK` 或 `nil` 一样。类型变量 `'a` 出现在 `list` 前面，这表示 `list` 本身并非一个类型，而是一个类型构造子（type constructor）：即它必须先应用于诸如 `int` 或 `string` 这样的类型参数之后，才能成为一个有意义的类型。类型构造子写在它们所应用的类型之后，因此 `int list` 是整数列表的类型。

构造子 `::`（发音为「cons」）携带一个类型为 `'a * 'a list` 的值；即一个对（pair），其中第一个元素类型为 `'a`，第二个元素类型为 `'a list`。例如，对于一个字符串列表，第一个元素为一个字符串，第二个元素则是另一个字符串列表（即该列表的「尾部」（tail））。

虽然 `'a` 可以代表任何类型——例如可以构造 `int list list`——但同一个列表中的元素必须全部为相同类型。如果确实需要在同一个列表中包含不同类型的对象，可以使用某个数据类型（datatype）作为列表类型构造子的参数，例如：`vehicle list` 就可以同时包含汽车（car）和卡车（truck）。

大多数数据结构都是不可变的（immutable）：它们不能通过赋值语句修改。例如，如果变量 `a` 持有前文所示描述 Fred 的六人座汽车（car）的记录值，那么既无法通过赋值语句修改该记录。例如，若变量 `a` 存储了上述描述 Fred 的 6 人座汽车的记录值，则它不能通过赋值语句被修改。

然而，这条规则存在一个例外。有一种特殊的数据类型 `ref`，表示可变引用（mutable references）：

```
datatype 'a ref = ref of 'a
```

这种内置数据类型具有“特殊”的性质，与预定义于 Standard ML 中的 `list` 数据类型不同——虽然 `list` 是预定义的，但它完全可以被用户自行实现，且并无特殊之处。

例如，若有如下记录：

```
{wheels=4, passengers=ref 6, owner="Fred"}
```

则该记录的 `passengers` 字段可以通过赋值修改为其他值。但注意，这个记录的类型是：

```
{wheels: int, passengers: int ref, owner: string}
```

这与 Fred 原本汽车的类型并不相同。

A.1. 表达式

ML 中的表达式可取如下形式：

- exp → id

  一个表达式（expression）可以是单个标识符（identifier）。ML 有两种类型的标识符：字母数字标识符（alphanumeric identifier）由字母和数字组成，并且以字母开头（与 Pascal 类似，但 ML 允许在字母数字标识符中包含下划线 `_` 和撇号 `'`）。符号标识符（symbolic identifier）则由字符 `!%&$+-/:<=>?@\~\^|#*'` 以任意组合构成。

  有些字母数字组合是保留字（reserved words）（如 `let` 和 `end`）；某些符号组合也同样是保留字（reserved words）（如 `|` 和 `=>`）。

- exp → exp id exp

  一个中缀运算符（infix operator）可以放在两个表达式（expressions）之间，例如 `a+b`、`(a+b)*c` 或 `a*b+c`。和 Pascal 或 C 一样，不同的运算符有不同的优先级（precedences）。不过在 ML 中，任何标识符（identifier）都可以被声明为带有特定优先级的中缀运算符，而像 `+` 和 `*` 这样的运算符也只是普通的标识符，只是默认被定义为中缀的而已。

- exp → (exp)

  一个表达式（expression）可以用括号括起来而不改变其含义；这与 Pascal 一样，可用于覆盖运算符的通常优先级（precedence）。

- exp → exp exp

  在 ML 中，函数应用（function application）通过先写函数、再跟随参数的方式表示。因此，`f x` 就是函数 `f` 应用到参数 `x` 上。如果你写成 `f(x)`，看起来更类似于 Pascal，但实际上括号并非必要，你同样可以写成 `(f)x`。

  一个表达式可以求值为一个函数；如果 `g y` 返回的结果是一个函数，那么这个函数可以通过 `(g y) x` 的形式应用到 `x` 上。但实际上，函数应用是左结合的，因此 `g y x` 会被解析为 `(g y) x`。另一方面，一个函数调用的参数也可能是另一个函数调用，例如 `f(g y)` 或者 `(g y)(h x)`。

  在 ML 中，每个函数都严格只接受一个参数。

- exp → id

- exp → id exp

- exp → exp id exp

  数据类型（datatype）的值既可以用一个不带值的常量数据构造子（data constructor）构造（上述三个规则中的第一个规则），也可以用一个携带值的构造子（上述第二个规则）应用于参数构造。如果一个携带值的构造子所携带的值是一个二元组（2-tuple），则可以将其声明为中缀（infix）形式；这种情况下，构造子写在它的两个参数之间，如上面第三条语法规则所示。

  列表构造子 `::` 就是这样一个例子：`3::nil` 是一个单元素列表，其中 `::` 应用于二元组`(3,nil)`。并且，`::` 被声明为右结合（right-associative）的，因此 `1::2::3::4::nil` 是一个包含四个整数的列表。

- exp → (exp, exp)

- exp → (exp, exp, exp)

  在 ML 中，可以通过用括号括起来的两个或多个用逗号分隔的表达式来构造n元组（n-tuples）。当元组表达式（tuple expression）求值时，一个元组形式的记录值被创建在内存中。例如：

  - `(3,"abc",7)` 是一个包含三个元素的记录，即一个3-元组（3-tuple）；
  - 这与 Lisp 中的 `cons` 或 Pascal 中的记录类似。

  记录的结构取决于括号内逗号的组合方式。例如，表达式 `(3,"a",7)` 是一个包含一个整数、一个字符串和一个整数的 3-元组类型；它与 `((3,"a"),7)` 并不同，后者是一个包含了一个2-元组（2-tuple）和一个整数的2-元组（2-tuple）。

  函数只能接受一个参数的限制看起来似乎很严格；但实际中人们通常向函数传递一个n-元组（n-tuple），例如 `f(x,y,z)`。这样写就看起来更像是函数接受了多个参数。

- exp → {exprow}

- exprow → id = exp

- exprow → exprow, id = exp

  一个记录（record）可通过在大括号内写出以逗号分隔的多个带有字段名的表达式来构造，这些字段名用来标识各个字段的值。

- exp → `let` dec `in` exp `end`

  一个 `let` 表达式（expression）引入了一个局部声明（declaration）（dec）。此声明所定义的名称仅在 `let` 和 `end` 之间可见。声明的具体形式参见 Section A.3。

- exp → `fn` pat => exp

  关键字 `fn` 发音为 lambda（λ），用于定义匿名函数（anonymous functions）。语法符号 `pat` 表示模式（pattern）；一个简单的模式示例就是单个标识符（但请参阅下文）。例如，表达式：

```
fn x => x+3
```

定义了一个将其参数加 3 的函数，因此表达式 `(fn x => x + 3) 7` 的值为 10。

Lambda 表达式可以嵌套，如：

```
fn x => fn y => x + y
```

这定义了一个返回函数的函数。ML 采用 词法作用域 (lexical (static) scope)，因此表达式：`(fn x => fn y => x + y) 3` 在任何上下文中的含义都与 `fn y => 3 + y` 完全相同。

嵌套的 lambda 表达式提供了一种定义多参数函数的替代方式，因此如果 f 是：

```
fn x => fn y => fn z => x + y + z
```

则可以通过 `f a b c` 的形式向函数 f 应用三个参数；在这种情况下，函数 f 称作柯里化 (curried) 函数。

实际上，`fn` 表达式的语法比上面的语法规则所暗示的更丰富。更精确地说： 允许使用形式为：

- `fn` match

  的表达式，其中 match 是一系列由竖线 (`|`) 分隔的规则 (rules)：

- match → rule
- match → match | rule
- rule → pat ⇒ exp

  当一个由多个规则 (rules) 定义的函数被应用于某个参数时，这些模式 (patterns) 将按顺序依次检查，直到找到第一个匹配成功的模式。然后，对应于该模式的表达式将被求值。

- exp → `case` exp `of` match

  一个 `case` 表达式会首先对其参数 exp 求值，然后根据所得值依次匹配 match 中的模式 (patterns)，找到第一个匹配成功的模式后，执行对应的绑定并对相应的右侧表达式求值。

A.2. 模式匹配

最简单的模式 (pattern) 就是一个变量（总是可以匹配），但也存在其他类型的模式：

- pat → constant

  当模式为一个整数、浮点数或字符串常量 c 时，它仅能匹配值 c。如果给定参数与 c 不相等，则匹配失败。注意，编译时的类型检查确保常量的类型（如 `int`）始终与参数的类型一致。

- pat → id

  当模式为一个 常量构造子 (constant constructor)（通过之前的 数据类型声明 (datatype declaration) 定义）时，它仅匹配该常量构造子。

  编译时的类型检查确保参数一定属于对应的数据类型。如果标识符并未通过数据类型声明定义为构造子，则它代表的是匹配中的变量，其行为与构造子截然不同（见下文）。这种情况经常会引起较大的混淆（并产生错误）。

  为了缓解这一问题，有一种常见的惯例是：构造子（不幸的是，除了 `nil`）以大写字母开头，而变量则不以大写字母开头。

- pat → id pat

  当模式为构造子应用于另一个模式 p 时，若参数是由同一构造子构建且 p 能匹配该构造子携带的值，则此模式匹配成功。因此，模式 `TRUCK 3.5` 能匹配由表达式 `TRUCK(2.5+1.0)` 所构造的值，但无法匹配值 `MOTORCYCLE` 或 `TRUCK(3.0)`。

- pat → (pat)

  pattern
  模式可以加上括号来消除语法上的歧义，这并不改变其含义。

- pat → (pat, pat)
- pat → (pat, pat, pat)

  n-tuple pattern
  一个 n-元组模式（对于任意 $n \geq 2$）当且仅当元组模式的每个元素都匹配参数中的对应元素时，才能匹配 n-元组参数。static type checking 静态类型检查确保参数是一个 n-元组（且 n 的值正确）。

- pat → {patrow}
- patrow → id = pat, patrow
- patrow →
- patrow → ...

  record pattern
  记录体模式是一个由 `id=pat` 组成的序列，其中 `id` 是记录体的标签。record

  tuple pattern
  与元组模式类似，记录体模式在所有字段都匹配参数的对应字段时才会匹配成功。记录体模式可以以省略号 `(...)` 结尾，此时参数中剩余的字段会自动匹配；但这些剩余字段的名称必须在 compile time 编译时 是可确定的。

- pat → _

  wild-card pattern
  下划线 `_` 是一个 通配模式 ，它可以匹配任何参数。

- pat → id

  pattern constructor
  当 模式 是一个标识符且未声明为 构造子 时，一个新的变量将被绑定到参数上，并且该匹配始终成立。因此，函数 `fn x => x + 3` 具有模式 `x`，它是一个变量。当该函数应用于某个参数时，`x` 会被实例化，并取该参数的值。变量 `x` 的作用域是整个函数的右侧部分（即 `x + 3`），因此该函数会对其参数加三。static type checking 静态类型检查 会确保该参数始终是一个整数。

  pair 2-tuple
  函数 `fn(a, b) => a + b * 2` 使用了一个对（即二元组）作为形式参数；该对的第一个元素会被绑定到变量 `a`，第二个元素会被绑定到变量 `b`。因此，尽管 ML 语言要求所有函数都必须有且仅有一个参数，但可以用这种方式模拟多参数函数，这种风格与传统方法非常相似。

  现在来看一个根据列表的不同长度作出不同处理的函数：

```
fn nil => 0
| x::nil => 1
| z => 2
```

  cons cell head tail
  第一条规则仅匹配空列表（`nil`）。第二条规则匹配的是构造单元，即列表的头部和尾部都必须匹配；由于 `x` 是一个变量，它可以匹配任意值，而尾部必须匹配空列表（`nil`），因此这条规则匹配的列表长度必须为1。最后一条规则会匹配任何未被前两条规则捕获的情况。

A.3. 声明

在 ML 中可以声明几种不同类的内容，每种都有不同的关键字用于引入声明：

- dec → `val` pat = exp

  val declaration pattern
  在 ML 中， `val` 声明 可以用于定义新变量，例如 `val x = 5`，这将 x 绑定到 5。在这种情况下， 模式 是一个简单的变量模式，它总是能够匹配。

`val` 声明的另一种用途是用于提取记录体（record）的字段，例如：

```
val (a, (b, c), d) = g(x)
```

在这种情况下，`g(x)` 返回的类型必须是一个 3 元组（3-tuple），其中第二个元素是一个对（pair）。3 元组的第一个和最后一个分量（component）分别被绑定到 a 和 d，而对的两个元素被绑定到 b 和 c。

最后，`val` 声明也可以用于可能无法匹配的情况，例如：

```
val TRUCK gross_weight = v
```

如果 v 这个 `vehicle` 确实是 `TRUCK`，那么变量 `gross_weight` 将被绑定到 `TRUCK` 构造器（constructor）携带的值。否则，`val` 声明会因引发 `Bind` 异常（exception）而失败。

- dec → `val rec` pat = exp

通常，由 `val` 声明（val declaration）绑定的标识符（identifier）的作用域（scope）不包括声明中的表达式（expression）`exp`。然而，关键字 `rec` 表示一个递归声明（recursive declaration）；在这种情况下，`exp` 必须以关键字 `fn` 开头。因此，以下是一个计算列表（list）长度的函数：

```
val rec length = fn nil => 0 | fn a::r => 1 + length(r)
```

- dec → `fun` clauses

为 `val rec` 声明提供语法糖（syntactic sugar）是很方便的。以下是一些示例，以及它们在不使用 `fun` 关键字时的等价形式：

```
fun length nil = 0
  | length (a::r) = 1 + length(r)
```

```
val rec length = fn nil => 0
                  | (a::r) => 1 + length(r)
```

```
fun f a b c = a + b + c
```

```
val rec f = fn a => fn b => fn c => a + b + c
```

需要注意的是，如果 `f` 不是递归的，使用 `rec` 关键字不会产生影响。

- dec → `datatype` datbind

数据类型（datatype）使用本附录开头描述的语法进行声明。

- dec → `type` type-binding

类型缩写（type abbreviation）用于声明一个新类型，该类型等价于某个已存在（可能是未命名的）类型。例如：

```
type intpair = int * int
type comparison = int * int -> bool
```

类型 `intpair` 现在与 `int × int` 相同，而 `comparison` 是“从整数对（pair of integers）到布尔值（Boolean）的函数（function）”的缩写。

类型缩写可以是参数化（parametrized）的，例如：

```
type 'a pair = 'a * 'a
type 'b predicate = 'b -> bool
```

其中，`'a pair` 表示元素类型为 `'a` 的成对数据结构，而 `'b predicate` 表示从 `'b` 类型映射到布尔值的谓词（predicate）。

- dec → dec

在允许使用单个声明的地方，也可以连续使用多个声明（declaration）。较早声明中绑定的标识符（identifier）可以在后续声明中使用。

- dec → **etc**.

还有几种其他类型的声明（declaration），但由于篇幅限制，这里无法详细说明。

A.4. 一些例子

为了说明 ML 的使用，我们将解释本书其他地方使用的一些示例。首先来看 countzeros 程序（第 83 页）：

```
fun count p = let fun f (a::r) = if p a then 1+f(r) else f r
                    | f nil = 0
              in f
              end

fun curry f x y = f(x,y)

val countzeros = count (curry (fn (w,z)=> w=z) 0)
```

函数 count 接受一个参数 p，并返回另一个函数作为结果。该返回的函数接受一个列表作为参数，并返回一个整数。更简单的理解方式是，count 是一个柯里化（curried）的双参数函数。无论如何，它的类型是：

```
count: ('a -> bool) -> (('a list) -> int)
count: ('a -> bool) -> 'a list -> int
```

类型被写了两次，第二次使用了最少数量的括号（注意 -> 运算符是右结合（associates to the right）的）。

编译器可以自动推导出 count 的类型（事实上，它可以推导出所有函数的类型）。在这个例子中，由于 p 被应用于某个参数，它必须是一个函数（function）；而由于 p 的返回值被用于 if 表达式中，p 必须返回一个布尔值（bool）。

第二个参数的写法是 a::r 和 nil，这意味着它必须是一个列表（list）；此外，由于 a 在表达式中被重复使用，这表明列表的元素类型必须与 p 的参数类型匹配。最后，由于 then 分支返回的表达式是 1 + count(r)，可以推导出 count 的返回类型是整数（int）。

那么 count 实际上做了什么呢？

参数 p 是一个谓词（predicate），它会被应用到列表的每个元素上。如果列表为空（即 count p nil），那么返回 0。否则，如果 p 在列表的第一个元素上返回 true，那么结果是 1 + count p 作用于列表的尾部；如果 p 返回 false，那么不会加 1。因此，count 计算的是列表中满足谓词 p 的元素个数。

curry 函数接受一个函数 f 作为参数。f 是一个接受二元组（2-tuple）作为参数的函数。而 curry f 的返回结果是一个函数 fn x => fn y => ...，它接受单个参数 x，然后返回一个新函数 fn y =>，这个新函数再接受一个参数 y。该函数的最终结果是 f(x, y)。因此，curry f 返回 f 的柯里化（curried）版本。

countzeros 函数用于计算整数列表中零的个数。它通过将 count 应用于合适的谓词（predicate）来实现这一点。当然，在 count 实际执行之前，它还需要被应用到另一个参数（一个列表），因此 countzeros 本身不会立即执行任何操作，直到它被应用到一个整数列表为止。

现在让我们来看 eq 函数（第 27 页）。这里我们为了简洁起见省略了一些情况：

```
fun eq(RECORD(a,i),RECORD(b,j)) =
      arbitrarily(i=j andalso eqlist(a,b), false)
| eq(INT i, INT j) = i=j
| eq(STRING a, STRING b) = arbitrarily(a=b, false)
| eq(ARRAY nil, ARRAY nil) = true
| eq(ARRAY(a::_), ARRAY(b::_)) = a=b
| eq(FUNC a, FUNC b) = raise Undefined
| eq(_,_) = false
and eqlist(a::al, b::bl) = eq(a,b) andalso eqlist(al,bl)
| eqlist(nil, nil) = true
```

`fun ... and` 语法允许定义 mutually recursive 互递归 的函数 `eq` 和 `eqlist`。函数 `eq` 给出了 pointer equality 指针相等 在一对 `dvalue` 参数上的语义（参见第 24 页），而 `eqlist` 模拟了 `dvalue` 列表上的 pointer equality 指针相等 。

调用 `eq` 的两个参数分别为 `x` 和 `y`。

- 如果它们都是 `RECORD`，那么 `x` 的 field-list 字段列表会绑定到 `a`，`y` 的字段列表会绑定到 `b`，它们的 offsets 偏移量分别绑定到 `i` 和 `j`。
- 然后，函数 `arbitrarily` 将以两个参数进行求值。

  第一个参数使用了 `andalso` 关键字，它执行 short-circuit 短路 布尔求值；等价于：

  ```
  if i=j then eqlist(a,b) else false
  ```

  第二个参数是 `false`。`arbitrarily` 仅返回其参数之一（参见第 3 页）。

  因此，在测试记录体 record 的相等性时，如果 `i = j`，则会调用 `eqlist` 函数，而 `eqlist` 可能会递归调用 `eq`。
- 如果其中一个参数是 `RECORD` 而另一个不是，则会匹配`(_,_)`这一模式，并返回 `false`。
- 如果两个参数都是 `STRING`，那么匹配第三个模式。
- 如果两个参数都是 `ARRAY`，那么会匹配第四个或第五个模式，这取决于它们的 element-lists 元素列表是否都为 `nil`。
- 如果两个参数都是 `FUNC`，那么匹配第六个模式，并执行 `raise Undefined`，抛出异常。在这种情况下，`eq` 不会返回任何结果，而是将控制权转交给最近的 exception handler 异常处理器 。

  函数 `eqlist` 具有两个匹配 clause 子句：

1. 一个匹配 nonempty lists 非空列表 ，它会测试 首元素的相等性，然后递归处理剩余部分。
2. 另一个匹配 empty lists 空列表 。

但是，如果两个列表的长度不同，那么递归调用最终会导致一个列表是 `nil` 而另一个不是。在这种情况下，没有任何模式可以匹配，运行时会抛出 runtime exception `Match` `Match` 异常 。

这个语义在 Section 3 中有详细说明。

```
functor CPSsemantics
  (structure CPS: CPS
   val minint: int
   val maxint: int
   val minreal: real
   val maxreal: real
   val string2real: string -> real
   eqtype loc
   val nextloc: loc -> loc
   val arbitrarily: 'a * 'a -> 'a
   type answer
   datatype dvalue =
     RECORD of dvalue list * int
   | INT of int
   | REAL of real
   | FUNC of dvalue list -> (loc * (loc -> dvalue) * (loc -> int)) -> answer
   | STRING of string
   | BYTEARRAY of loc list
   | ARRAY of loc list
   | UARRAY of loc list
   val handler_ref: loc
   val overflow_exn: dvalue
   val div_exn: dvalue):
sig
  val eval: CPS.var list * CPS.cexp
            -> dvalue list
            -> (loc * (loc -> dvalue) * (loc -> int))
            -> answer
end =
struct
  type store = loc * (loc -> dvalue) * (loc -> int)
  fun fetch ((_, f, _): store) (l: loc) = f l
  fun upd ((n, f, g): store, l: loc, v: dvalue) =
    (n, fn i => if i = l then v else f i, g)
  fun fetchi ((_, _, g): store) (l: loc) = g l
  fun updi ((n, f, g): store, l: loc, v: int) =
    (n, f, fn i => if i = l then v else g i)

  exception Undefined

  fun eq (RECORD (a, i), RECORD (b, j)) =
        arbitrarily (i = j andalso eqlist (a, b), false)
    | eq (INT i, INT j) = i = j
    | eq (REAL a, REAL b) =
        arbitrarily (a = b, false)
    | eq (STRING a, STRING b) =
        arbitrarily (a = b, false)
    | eq (BYTEARRAY nil, BYTEARRAY nil) = true
    | eq (BYTEARRAY (a :: _), BYTEARRAY (b :: _)) = a = b
    | eq (ARRAY nil, ARRAY nil) = true
    | eq (ARRAY (a :: _), ARRAY (b :: _)) = a = b
    | eq (UARRAY nil, UARRAY nil) = true
    | eq (UARRAY (a :: _), UARRAY (b :: _)) = a = b
    | eq (FUNC a, FUNC b) = raise Undefined
    | eq (_, _) = false
  and eqlist (a :: al, b :: bl) =
        eq (a, b) andalso eqlist (al, bl)
    | eqlist (nil, nil) = true
```

```
fun do_raise exn s =
  let val FUNC f = fetch s handler_ref
  in f [exn] s
  end

fun overflow (n: unit -> int, c: dvalue list -> store -> answer) =
  if (n () >= minint andalso n () <= maxint) handle Overflow => false then
    c [INT (n ())]
  else
    do_raise overflow_exn

fun overflowr (n, c) =
  if (n () >= minreal andalso n () <= maxreal) handle Overflow => false then
    c [REAL (n ())]
  else
    do_raise overflow_exn

fun evalprim
      ( CPS.+ : CPS.primop
      , [INT i, INT j]: dvalue list
      , [c]: (dvalue list -> store -> answer) list
      ) =
      overflow (fn () => i + j, c)
  | evalprim (CPS.-, [INT i, INT j], [c]) =
      overflow (fn () => i - j, c)
  | evalprim (CPS.*, [INT i, INT j], [c]) =
      overflow (fn () => i * j, c)
  | evalprim (CPS.div, [INT i, INT 0], [c]) = do_raise div_exn
  | evalprim (CPS.div, [INT i, INT j], [c]) =
      overflow (fn () => i div j, c)
  | evalprim (CPS.~, [INT i], [c]) =
      overflow (fn () => 0 - i, c)
  | evalprim (CPS.<, [INT i, INT j], [t, f]) =
      if i < j then t [] else f []
  | evalprim (CPS.<=, [INT i, INT j], [t, f]) =
      if j < i then f [] else t []
  | evalprim (CPS.>, [INT i, INT j], [t, f]) =
      if j < i then t [] else f []
  | evalprim (CPS.>=, [INT i, INT j], [t, f]) =
      if i < j then f [] else t []
  | evalprim (CPS.ieql, [a, b], [t, f]) =
      if eq (a, b) then t [] else f []
  | evalprim (CPS.ineq, [a, b], [t, f]) =
      if eq (a, b) then f [] else t []
  | evalprim (CPS.rangechk, [INT i, INT j], [t, f]) =
      if j < 0 then if i < 0 then if i < j then t [] else f [] else t []
      else if i < 0 then f []
      else if i < j then t []
      else f []
  | evalprim (CPS.boxed, [INT _], [t, f]) = f []
  | evalprim (CPS.boxed, [RECORD _], [t, f]) = t []
  | evalprim (CPS.boxed, [STRING _], [t, f]) = t []
  | evalprim (CPS.boxed, [ARRAY _], [t, f]) = t []
  | evalprim (CPS.boxed, [UARRAY _], [t, f]) = t []
  | evalprim (CPS.boxed, [BYTEARRAY _], [t, f]) = t []
  | evalprim (CPS.boxed, [FUNC _], [t, f]) = t []
  | evalprim (CPS.!, [a], [c]) =
      evalprim (CPS.subscript, [a, INT 0], [c])
  | evalprim (CPS.subscript, [ARRAY a, INT n], [c]) =
      (fn s => c [fetch s (nth (a, n))] s)
  | evalprim (CPS.subscript, [UARRAY a, INT n], [c]) =
      (fn s => c [INT (fetchi s (nth (a, n)))] s)
  | evalprim (CPS.subscript, [RECORD (a, i), INT j], [c]) =
```

```
      c [nth (a, i + j)]
  | evalprim (CPS.ordof, [STRING a, INT i], [c]) =
      c [INT (String.ordof (a, i))]
  | evalprim (CPS.ordof, [BYTEARRAY a, INT i], [c]) =
      (fn s => c [INT (fetchi s (nth (a, i)))] s)
  | evalprim (CPS.:=, [a, v], [c]) =
      evalprim (CPS.update, [a, INT 0, v], [c])
  | evalprim (CPS.update, [ARRAY a, INT n, v], [c]) =
      (fn s => c [] (upd (s, nth (a, n), v)))
  | evalprim (CPS.update, [UARRAY a, INT n, INT v], [c]) =
      (fn s => c [] (updi (s, nth (a, n), v)))
  | evalprim (CPS.unboxedassign, [a, v], [c]) =
      evalprim (CPS.unboxedupdate, [a, INT 0, v], [c])
  | evalprim (CPS.unboxedupdate, [ARRAY a, INT n, INT v], [c]) =
      (fn s => c [] (upd (s, nth (a, n), INT v)))
  | evalprim (CPS.unboxedupdate, [UARRAY a, INT n, INT v], [c]) =
      (fn s => c [] (updi (s, nth (a, n), v)))
  | evalprim (CPS.store, [BYTEARRAY a, INT i, INT v], [c]) =
      if v < 0 orelse v >= 256 then raise Undefined
      else (fn s => c [] (updi (s, nth (a, i), v)))
  | evalprim (CPS.makeref, [v], [c]) =
      (fn (l, f, g) => c [ARRAY [l]] (upd ((nextloc l, f, g), l, v)))
  | evalprim (CPS.makerefunboxed, [INT v], [c]) =
      (fn (l, f, g) => c [UARRAY [l]] (updi ((nextloc l, f, g), l, v)))
  | evalprim (CPS.alength, [ARRAY a], [c]) =
      c [INT (List.length a)]
  | evalprim (CPS.alength, [UARRAY a], [c]) =
      c [INT (List.length a)]
  | evalprim (CPS.slength, [BYTEARRAY a], [c]) =
      c [INT (List.length a)]
  | evalprim (CPS.slength, [STRING a], [c]) =
      c [INT (String.size a)]
  | evalprim (CPS.gethdlr, [], [c]) =
      (fn s => c [fetch s handler_ref] s)
  | evalprim (CPS.sethdlr, [h], [c]) =
      (fn s => c [] (upd (s, handler_ref, h)))
  | evalprim (CPS.fadd, [REAL a, REAL b], [c]) =
      overflowr (fn () => a + b, c)
  | evalprim (CPS.fsub, [REAL a, REAL b], [c]) =
      overflowr (fn () => a - b, c)
  | evalprim (CPS.fmul, [REAL a, REAL b], [c]) =
      overflowr (fn () => a * b, c)
  | evalprim (CPS.fdiv, [REAL a, REAL 0.0], [c]) = do_raise div_exn
  | evalprim (CPS.fdiv, [REAL a, REAL b], [c]) =
      overflowr (fn () => a / b, c)
  | evalprim (CPS.feql, [REAL a, REAL b], [t, f]) =
      if a = b then t [] else f []
  | evalprim (CPS.fneq, [REAL a, REAL b], [t, f]) =
      if a = b then f [] else t []
  | evalprim (CPS.flt, [REAL i, REAL j], [t, f]) =
      if i < j then t [] else f []
  | evalprim (CPS.fle, [REAL i, REAL j], [t, f]) =
      if j < i then f [] else t []
  | evalprim (CPS.fgt, [REAL i, REAL j], [t, f]) =
      if j < i then t [] else f []
  | evalprim (CPS.fge, [REAL i, REAL j], [t, f]) =
      if i < j then f [] else t []

type env = CPS.var -> dvalue

fun V env (CPS.INT i) = INT i
  | V env (CPS.REAL r) =
      REAL (string2real r)
  | V env (CPS.STRING s) = STRING s
```

```
    | V env (CPS.VAR v) = env v
    | V env (CPS.LABEL v) = env v

  fun bind (env: env, v: CPS.var, d) =
    fn w => if v = w then d else env w

  fun bindn (env, v :: vl, d :: dl) =
        bindn (bind (env, v, d), vl, dl)
    | bindn (env, nil, nil) = env

  fun F (x, CPS.OFFp 0) = x
    | F (RECORD (l, i), CPS.OFFp j) =
      RECORD (l, i + j)
    | F (RECORD (l, i), CPS.SELp (j, p)) =
      F (nth (l, i + j), p)

  fun E (CPS.SELECT (i, v, w, e)) env =
      let val RECORD (l, j) = V env v
      in E e (bind (env, w, nth (l, i + j)))
      end
    | E (CPS.OFFSET (i, v, w, e)) env =
      let val RECORD (l, j) = V env v
      in E e (bind (env, w, RECORD (l, i + j)))
      end
    | E (CPS.APP (f, vl)) env =
      let val FUNC g = V env f
      in g (map (V env) vl)
      end
    | E (CPS.RECORD (vl, w, e)) env =
      E e (bind (env, w, RECORD (map (fn (x, p) => F (V env x, p)) vl, 0)))
    | E (CPS.SWITCH (v, el)) env =
      let val INT i = V env v
      in E (nth (el, i)) env
      end
    | E (CPS.PRIMOP (p, vl, wl, el)) env =
      evalprim
        ( p
        , map (V env) vl
        , map (fn e => fn al => E e (bindn (env, wl, al))) el
        )
    | E (CPS.FIX (fl, e)) env =
      let
        fun h r1 (f, vl, b) =
          FUNC (fn al => E b (bindn (g r1, vl, al)))
        and g r =
          bindn (r, map #1 fl, map (h r) fl)
      in
        E e (g env)
      end

  val env0 = fn x => raise Undefined

  fun eval (vl, e) dl =
    E e (bindn (env0, vl, dl))
end
```

Standard ML of New Jersey 是许多人共同完成的工作（参见 Acknowledgement）。该系统由普林斯顿大学和 AT&T 贝尔实验室联合开发，并有来自其他机构的贡献。

Standard ML of New Jersey 可以从 http://smlnj.org 获取。

Bibliography

[1] A. Appel, *Compiling with Continuations*. Cambridge University Press, 1992. [Online]. Available: https://books.google.com.tw/books?id=0Uoecu9ju4AC

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proc. 16th ACM Symp. on Principles of Programming Languages*, New York: ACM Press, 1989, pp. 25–35.

[3] R. Kelsey and P. Hudak, "Realistic Compilation by Program Transformation," in *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages*, New York: ACM Press, 1989, pp. 281–292.

[4] M. J. C. Gordon, A. J. R. G. Milner, L. Morris, M. C. Newey, and C. P. Wadsworth, "A metalanguage for interactive proof in LCF," in *Fifth ACM Symp. on Principles of Programming Languages*, New York: ACM Press, 1978.

[5] L. Cardelli, "Compiling a Functional Language," in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, ACM Press, New York, 1984, pp. 208–217.

[6] R. Milner, "A Proposal for Standard ML," in *ACM Symposium on LISP and Functional Programming*, New York: ACM Press, 1984, pp. 184–197.

[7] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*. Cambridge, MA: MIT Press, 1989.

[8] A. W. Appel and D. B. MacQueen, "A Standard ML Compiler," in *Functional Programming Languages and Computer Architecture (LNCS 274)*, G. Kahn, Ed., New York: Springer-Verlag, 1987, pp. 301–324.

[9] D. C. J. Matthews, "Papers on Poly/ML," Feb. 1989.

[10] J. Welsh, W. J. Sneeringer, and C. A. R. Hoare, "Ambiguities and Insecurities in Pascal," *Software—Practice and Experience*, vol. 7, no. 6, pp. 685–696, 1977.

[11] J. A. Rees, W. D. Clinger, and others, "Revised^3 Report on the Algorithmic Language Scheme", *SIGPLAN Notices*, vol. 21, no. 12, pp. 37–79, Dec. 1986.

[12] C. Reade, *Elements of Functional Programming*. Reading, MA: Addison-Wesley, 1989.

[13] P. J. Landin, "The mechanical evaluation of expressions," *Computer Journal*, vol. 6, no. 4, pp. 308–320, 1964.

[14] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge, MA: MIT Press, 1977.

[15] M. J. C. Gordon, *The Denotational Description of Programming Languages*. New York: Springer-Verlag, 1979.

[16] D. A. Schmidt, *Denotational Semantics: A Methodology for Language Development*. Boston: Allyn, Bacon, 1986.

[17] L. C. Paulson, *ML for the Working Programmer*. Cambridge: Cambridge University Press, 1992.

索引